WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
PLURALITY.USPT.	1425438
PLURALITIES.USPT.	12476
PLURALITYS.USPT.	2
EXECUTION.USPT.	98159
EXECUTIONS.USPT.	4672
FUNCTIONAL.USPT.	315207
FUNCTIONALS.USPT.	127
UNIT$	0
UNIT.USPT.	1031882
UNITA.USPT.	32
UNITAACK.USPT.	2
(L10 AND (PLURALITY WITH ((EXECUTION OR FUNCTIONAL) ADJ1 UNIT$))).USPT.	12

There are more results than shown above. Click here to view the entire set.

Database:
- US Patents Full-Text Database
- US Pre-Grant Publication Full-Text Database
- JPO Abstracts Database
- EPO Abstracts Database
- Derwent World Patents Index
- IBM Technical Disclosure Bulletins

Search: L11

Refine Search

Recall Text | Clear

Search History

DATE: Friday, April 04, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
	DB=USPT; PLUR=YES; OP=ADJ		
L11	L10 and (plurality with ((execution or functional) adj1 unit$))	12	L11
L10	l1 and ((global adj1 register$) and (local adj1 register$))	52	L10
L9	l1 and (associated with global with local with register$)	12	L9
L8	(each with associated with global with local with register$)	0	L8
L7	l1 and (each with associated with global with local with register$)	0	L7
L6	(register adj1 file$) same (partition$) same ((global or local) adj1 (register$))	4	L6
L5	(register adj1 file$) with (partition$) with ((global or local) adj1 (register$))	2	L5
L4	L2 and ((global adj1 register$) and (local adj register$))	7	L4
L3	L1 and ((register adj1 file) with (partition$ or programmable))	101	L3
L2	L1 and ((register adj1 file) same (partition$ or programmable))	208	L2
L1	((712/$)!.CCLS.)	7901	L1

END OF SEARCH HISTORY

L11: Entry 10 of 12 File: USPT May 5, 1998

DOCUMENT-IDENTIFIER: US 5748950 A
TITLE: Method and apparatus for providing an optimized compare-and-branch instruction

Detailed Description Text (10):
Within the processor illustrated in FIG. 2, all operations take place at the register level. Source operands specify either a global register, a local register or a constant value as instruction operands. The functional units are coupled to the register file 30 via three independent 32-bit buses. These are identified as source 1 (SRC1), source 2 (SRC2) and the destination (DEST) buses. In alternative embodiments of the present invention, a wider single bus, or smaller multiplexed common bus may be utilized (or various combinations of separate buses) for communicating between the register file 30 and the various functional units.

Detailed Description Text (20):
The first instruction at address A.sub.-- 0 is a compare-and-branch instruction, particularly a `compare ordinal and branch if equal` instruction. The values stored in global register 5 and local register 5 are compared, and if they are equal, the next instruction to be executed will be at the A.sub.-- targ address. If the g5 and r5 registers do not contain equal ordinal values, then the next instruction executed is the instruction at address A.sub.-- 0+4. The instruction at the A.sub.-- 0+4 address is an add ordinal instruction for adding the values in global registers 8 and 9 and putting the result in global register 10. The instruction at the A.sub.-- targ address is a subtract ordinal instruction which subtracts the value in local register 8 from local register 9 and stores the result in global register 11.

Current US Original Classification (1):
712/245

Current US Cross Reference Classification (1):
712/41

CLAIMS:

1. A data processing unit for executing a composite instruction having a first opcode portion corresponding to a first instruction, said data processing unit comprising:

an instruction memory for storing said composite instruction;

an instruction bus coupled to said instruction memory for conveying said composite instruction;

an instruction sequencer coupled to said instruction bus for receiving said composite instruction from said instruction memory; and

a plurality of execution units coupled to said instruction sequencer for executing data processing unit instructions,

said instruction sequencer, in response to receiving said composite instruction, translating the first opcode portion into a second opcode portion corresponding to a second instruction.

7. A computer system comprising:

an input/output (I/O) means for providing a communication interface;

a memory means coupled to the I/O means for storing instructions and computer data;

data input means coupled to the I/O means for providing data input and data output to interface the computer system with a computer user; and

microprocessor means coupled to the I/O means for executing a composite instruction wherein said composite instruction includes a first opcode portion corresponding to a first instruction, said microprocessor means comprising:

an instruction memory for storing said composite instruction;

an instruction bus coupled to said instruction memory for conveying said composite instruction;

an instruction sequencer coupled to said instruction bus for receiving said composite instruction from said instruction memory; and

a plurality of execution units coupled to said instruction sequencer for executing instructions,

said instruction sequencer in response to receiving said composite instruction, translating the first opcode portion into a second opcode portion corresponding to a second instruction.

L11: Entry 12 of 12 File: USPT May 15, 1990

DOCUMENT-IDENTIFIER: US 4926323 A
TITLE: Streamlined instruction processor

Detailed Description Text (16):
The execution unit 24 also includes a plurality of special purpose registers shown conceptually as a block, 54, which provide controls and data for prespecified processor operations. A specification of the special purpose registers and their contents is provided in the cross-referenced Am29000 User's Manual. It is not set out in detail here. Special purpose registers pertinent to the present invention are described where appropriate.

Detailed Description Text (100):
As illustrated in FIG. 9, register numbers 0 and 1 store special information, register numbers 64-127 are global registers, and register numbers 128-255 are local registers. Register numbers 2-63 are not implemented.

Detailed Description Text (102):
Global register 1 contains the stack pointer, which is used in the addressing of local registers as explained below.

Detailed Description Text (103):
Global registers 64-127 are accessed with the 7 least significant bits of the register number from the subject instruction. Local registers 128-255, indicated when the most significant bit of the register number is 1, are addressed by adding bits 8-2 of the stack pointer to the 8-bit register number and truncating the result to 7 bits. The most significant bit of the original register number is left at 1.

Detailed Description Text (104):
The stack pointer is a 32-bit register that may be an operand of an instruction as any other general purpose register. However, a shadow copy of global register 1 is maintained by processor hardware to be used in local register addressing. This shadow copy is set with the results of arithmetic and logic instructions. If the stack pointer is set with the result of any other instruction class, local registers cannot be accessed predictably until the stack pointer is once again set with an arithmetic or logical instruction. A modification of the stack pointer has a delayed effect on the addressing of the local registers. An instruction which writes to the stack pointer or indirect pointer can be immediately followed by an instruction which reads the stack pointer or indirect pointer. However, any instruction which references a local register also uses the value of the stack pointer or indirect pointer to calculate an absolute register number. Because of the pipeline implementation, at least one cycle of delay must separate an instruction which updates the stack pointer or indirect pointer and an instruction which references a local register. In most systems, this affects procedure call and return type instructions only. In general, though, an instruction which immediately follows a change to the stack pointer or indirect pointer should not reference a local register. Note that this restriction does not apply to a reference of a local register via an indirect pointer that has not been subject of an update.

Detailed Description Text (107):
The indirect pointers are implemented as three special purpose registers (block 54 of FIG. 2), indirect pointer C, indirect pointer A and indirect pointer B. Indirect pointer C is an unprotected special purpose register which provides a register number for use as the C port for writes to the register file when an instruction specifies the indirect pointer by call to global register 0 in its RC field. Likewise, indirect pointer A is an unprotected special purpose register which

provides a register number for the A port for reads from the register file when an instruction RA field has the value of 0 specifying global register 0. Indirect pointer B is an unprotected special purpose register which provides a register number for the B port reads to the register file when the instruction has an RB field specifying global register 0.

Detailed Description Text (108):
The register address generator 43 is illustrated in more detail in FIG. 8. It includes a stack pointer register 806 coupled to receive bits 8-2 of global register 1 whenever global register 1 is written across the R bus with the result of an arithmetic or logic instruction. In addition, the register address generator 43 includes indirect pointer register A, 807, indirect pointer register B, 808, and indirect pointer register C, 809.

Detailed Description Text (114):
The channel control register is a protected special purpose register used to report exceptions during external accesses. It is also used to re-start interrupted load-multiple and store-multiple operations and to re-start other external accesses when possible, for instance, when translation lookaside buffer misses are serviced. The channel control register is updated on the execution of every load or store instruction, and on every load or store in a load multiple or store multiple sequence, except under special control conditions indicated by the current processor status register. This channel control register includes control fields indicating the nature of the load or store operation and a target register TR field indicating the absolute register number of a data operand for the current transaction, either a load target or store data source. The register number in this field is absolute and reflects the stack pointer addition when the indicated register is a local register. It also includes a not needed NN bit indicating that even though the channel control register contains valid information for an uncompleted load operation, data requested by that uncompleted operation is not needed. This situation arises when a load instruction is overlapped with an instruction which writes to the target register in the register file of the load. Other fields of the channel control register are described in detail in the Am29000 User's Manual.

Detailed Description Text (121):
As illustrated in FIG. 10, the general purpose registers in the register file 40 are partitioned into 16 banks, each including 16 registers; however, lower banks are unimplemented in the system described in the Am29000 User's Manual. Register positions 0 and 1 are not included in bank 0 as they are not, in fact, general purpose registers. A special purpose register in the special purpose register file 54 holds 16 register bank protection bits as indicated in the left hand column of FIG. 10. One bit corresponds to each of the 16 banks of general purpose registers. When the register protection bit for a given bank is 1, attempts accesses to that register as indicated by the register number on line 828 at the output of A multiplexer 814, line 827 at the output of B multiplexer 820, or line 824 at the output of C multiplexer 823, then a register bank protection trap is asserted by the processor. This gives the programmer the ability to restrict access to banks of registers in the register file 40. Register bank protection works only in the user mode and has no effect in the supervisor mode. Note that the protection is based on absolute register numbers; and in the case of local registers, stack pointer addition is performed before protection checking.

Detailed Description Text (123):
Register bank protection is particularly useful for supporting fast context switching in a multitasking processor. By protecting local registers assigned to multiple tasks, task switching time is minimized at the possible expense of an increase in procedure call and procedure execution time. Even so, this trade-off is appropriate in many real time applications.

Detailed Description Text (124):
The 128 local registers, absolute register numbers 128-255, may be partitioned into 8 banks of 16 registers each, with each bank of registers allocated to one of eight tasks resident in the processor. Partitioning can be made transparent to resident tasks because the stack pointer can be set so that the first register in each bank is addressed as local register 0. Even when two or more banks of local registers are

mounted into larger banks, each of the larger banks can still start with local register 0 combined with its unique stack pointer.

Detailed Description Text (125):
Since the stack pointer does not affect addressing of global registers, that is, absolute register numbers 2-127, global registers cannot be partitioned among multiple tasks. A task cannot be made to reference the proper global registers unless the registers allocated to a given task are known before execution, but this restriction is too severe in most cases. Global registers should be used, however, to maintain processor state information for the multiple tasks. More details of the context switching operation are provided in the crossreferenced Am29000 User's Manual.

Current US Original Classification (1):
712/238

Current US Cross Reference Classification (1):
712/207

DOCUMENT-IDENTIFIER: US 6463527 B1
TITLE: Spawn-join instruction set architecture for providing explicit multithreading

Detailed Description Text (10):
In the example program above, Rn threads are indexed j, j+1, . . . j+Rn-1. The command assigns REGS physical registers to local virtual registers. Typically initialized to 0, global register Rb is a base register for the SUMI command of the matching Join instruction. This Spawn and Join syntax is not too different than the use of similar symbols in the high-level language "FORK," described for example in the article by C. W. Kessler and H. Seidl, "The Fork95 Parallel Programming Language: Design, Implementation, Application," International Journal on Parallel Programming, 25(1), pp. 17-50 (1997), which is incorporated herein by reference in its entirety. The assembly code also follows the style of MIPS assembly code disclosed by Patterson and Hennessy in "Computer Organization & Design. The Hardware/Software Interface," 1994, which is incorporated by reference in its entirety.

Detailed Description Text (14):
A register file 30 containing a plurality of local and global registers (R1, R2, R3 . . . R64) is provided for use by TCUs 34. A prefix-sum unit 32, coupled to TCUs 34, is also provided for providing a hardware implemented prefix-sum calculation handling competing TCUs (as will be described in more detail below). In the preferred embodiment, the prefix-sum unit 32 is implemented in accordance with the disclosures in U.S. patent application Ser. No. 08/667,554 of June 1996 and continuation-in-part application Ser. No. 08/757,604, filed Nov. 29, 1996, the disclosures of which are both incorporated herein by reference. (It should be readily apparent, however, that any hardware or software implementation of the prefix-sum calculations described herein may be employed. Preferably, the implemented calculation can be performed with minimal delay. In this preferred embodiment, for example, the prefix-sum calculation is assumed to be performed in a single instruction cycle, as implemented in the above-identified patent applications.)

Detailed Description Text (28):
Using this format, the Join instruction contains a summation instruction that accumulates the number of threads reaching the Join command. In particular, each thread increments global register Rb. Once the value of Rb reaches n, the Spawn-Join loop is finished and the main program transitions from the parallel state to the serial state. The Join command preferably utilizes a parallel prefix sum computation (or using, for example, a "SUMI" instruction) with respect to variable Rb possibly using prefix-sum unit 32. The SUMI (for summing integers) command has a syntax: SUMI (Rb, Imm). A sequence of such commands with the same Rb causes summation of the immediate values Imm to be produced in parallel. At compile time, the relation 0.ltoreq.Imm.ltoreq.3 is inserted and the sequence takes unit time for.ltoreq.k instructions.

Detailed Description Text (34):
Each individual TCU 34a, 34b, 34c, . . . 34k preferably executes the Spawn-Join instructions in its thread serially tracking each instruction with a local program counter (PC), as is well known in the art. In an alternative embodiment, however, parallel architectures such as those based on superscalar (e.g., branch prediction, out-of-order execution, etc.), Very Long Instruction Word (VLIW), vectoring or any other parallel processing-type architecture known, may be employed to execute the Spawn-Join instructions in parallel to provide a state of "intrathread" parallelism. The TCUs 34 may perform a variety of functions such as global read, global write, as

well as local read and writes to registers in register file 30. This is done using functional units in a manner well known in the art. Although conflicts with concurrent reads of global registers (e.g., R64) can be avoided (e.g., when implementing a prefix-sum function), concurrent global writes must be synchronized using a prefix-sum functional unit in order to avoid serializing. When a concurrent write into a global register occurs, a prefix-sum unit calculates the prefix sums based on outputs from the relevant TCUs 34. The resulting prefix sums will award one of the TCUs 34 with the "right" to access the global register and guide the remaining TCUs 34 to proceed with their next instruction(s). In the alternative, a "Mark" instruction, which is a simpler or degenerate form of prefix-sum calculation will also be useful to designate the awarded TCU 34.

Detailed Description Text (36):
In the preferred embodiment, the processing elements making up TCUs 34a through 34k incorporate local instruction memory units 42a through 42k, respectively, as shown in FIG. 4. In a preferred construction of the computer architecture, instruction memory units 42a-42k store and/or track instructions that are to be performed by one of a plurality of groups 84 of standard functional units. Each group 84 preferably has a plurality of functional units 86, 87, 88, 89, etc. Additional functional units. Each functional unit is capable of executing instructions from one or more of the threads sent from tracking 42a-42k over bus 46a-46k, respectively, or any other conductive path known to those of ordinary skill in the art.

Detailed Description Text (38):
In an additional preferred construction local instruction memory units 52a through 52k store and/or track other instruction that are to be performed by on of tS0 a plurality of groups 94 of multi-operand functional units as shown in FIG. 6. Each group 94 preferably has a plurality of multi-operand functional units 96, 97, 98 etc. Each of the functional units is capable of executing multi-operand operations; the operands for each operations can come from different threads each having the an individual instruction (such as individual prefix-sum). All instruction referring concurrently to the same functional units must have the same base register. This provides inter-thread parallelism. A functional unit e.g., prefix-sum) can also get all its operands from a single thread, providing "intra-thread parallelism." Instructions from the threads are sent from tracking 52a-k over bus 56a-56k, or any other interconnect known to those of ordinary skill in the art.

Detailed Description Text (55):
An ability to "put on hold" threads and their enabled instructions by way of moving the registers of the threads and local variables to lower levels of the memory hierarchy can be provided. This ability is needed if higher levels of the memory hierarchy cannot hold all the memory they need. To the extent that local variables are used, the system will handle them similarly to local registers notwithstanding that each data type (e.g., integers, floating-points, structures, etc.) are treated separately.

Detailed Description Text (73):
The load-immediate command (li) initializes R1 to 0. The load-word command (lw) loads n into R2. The SPAWN command spawns R2 threads, indexed 0 to R2-1, and using 4 local registers per thread. The JOIN instruction matching the SPAWN instruction will count terminating threads into R3. R0$ always includes the thread index $, and will be a read-only local register. B_OFF is the base address for array B. The new load-word-array instruction is used to directly accessing array addresses. If R3$ equals 1, a prefix-sum is performed incrementing the counter R1. R1$ will provide the address into which to copy A($). A($) is then copied into compacted array D using the lwa and store-word-array (swa) instructions. Each thread reaching the JOIN command causes R3 to be incremented by 1 using a new parallel-sum integer instruction, which is part of the JOIN instruction. Once R3 becomes equal to R2, all of the threads have terminated and the program switches back to the serial state. The size of the compacted array is then stored into address Rn.

Current US Original Classification (1):
712/245

Current US Cross Reference Classification (4):

712/219

Current US Cross Reference Classification (5):
712/228

Current US Cross Reference Classification (6):
712/242

CLAIMS:

12. The computer system of claim 11, further comprising a plurality of local and global registers used by said thread control units during execution of the spawn-join instructions.

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L12	L1 and ((register adj1 file) with partition$ with programmable)	8	L12
L11	L10 and (plurality with ((execution or functional) adj1 unit$))	12	L11
L10	l1 and ((global adj1 register$) and (local adj1 register$))	52	L10
L9	l1 and (associated with global with local with register$)	12	L9
L8	(each with associated with global with local with register$)	0	L8
L7	l1 and (each with associated with global with local with register$)	0	L7
L6	(register adj1 file$) same (partition$) same ((global or local) adj1 (register$))	4	L6
L5	(register adj1 file$) with (partition$) with ((global or local) adj1 (register$))	2	L5
L4	L2 and ((global adj1 register$) and (local adj register$))	7	L4
L3	L1 and ((register adj1 file) with (partition$ or programmable))	101	L3
L2	L1 and ((register adj1 file) same (partition$ or programmable))	208	L2
L1	((712/$)!.CCLS.)	7901	L1

END OF SEARCH HISTORY

L9: Entry 11 of 12 File: USPT Sep 5, 1995

DOCUMENT-IDENTIFIER: US 5448707 A
TITLE: Mechanism to protect data saved on a local register cache during inter-subsystem calls and returns

Detailed Description Text (28):
The register file (RF) consists of 16 global registers and 4 floating-point registers that are preserved across procedure boundaries, and multiple sets of 16 local (or frame) registers that are associated with each stack frame. At any point in time, one can address thirty-two 32-bit registers, and four 80-bit floating-point registers (the 32 registers can also be used to hold floating-point values). Of the 32 registers, 16 are global registers and 16 are local registers. The difference is that the 16 global registers are unaffected when crossing procedure boundaries, i.e., they behave like "normal" registers in other processors. The local registers are affected by the call and return instructions.

Current US Original Classification (1):
712/228